AI Agent Final Report

Jacob Lerner and Matthew Barg

January 4, 2024

1 Our Bot - Patient Paul

1.1 Motivation

In developing our MiniMax-based game-playing bot, our initial considerations stemmed from observing the limitations of other approaches and strategizing to take advantage of them. We predict many other bots will tend to adopt overly aggressive tactics, and we identified an opportunity to exploit this tendency. By adopting a defensive stance, we aimed to create a strategy that could effectively counter aggressive opponents.

This idea gave us the impression that a solid defensive strategy may act as a solid starting point, taking advantage of opponents' rash or impatience and waiting for them to make crucial errors like boxing themselves in or placing themselves in awkward situations.

Our main mathematical decision to use the MiniMax algorithm in addition with iterative deepening is a reflection of our strategic flexibility and depth. Our bot can investigate the possible outcomes of different actions by utilizing MiniMax, a fundamental algorithm seen in class. This allows it to anticipate the opponent's response and optimize its own decisions accordingly. By gradually narrowing the search space and striking a balance between computing efficiency and strategic foresight, iterative deepening improves this process.

Our game theory is based on the ideas of controlling open areas and deftly avoiding confrontations with adversaries. The focus on defensive manoeuvres fits with our view that waiting for opponents to make mistakes and applying a patient approach can be a very effective tactic. Our bot prioritizes free space in an attempt to get a significant positioning advantage and possibly force opponents into unfavourable positions

1.2 Our Design

1.2.1 Evaluation Function - Movement

In designing our robot we knew the most important part would be our evaluation function. This would be the main differentiating factor to all other groups. We realised a depth limiting search for mini max would likely be the most common, so the evaluation function is how we would have to exploit different opponents. We have two heuristics with equal weight:

(1) How many walls are surrounding the position

(2) How many squares are accessible around the spot.

We intentionally chose a simple heuristic as we noticed that by having several components that affect where our player should move will in turn dilute the importance of each value and confuse Patient Paul. the first heuristic was achieved with a simple for loop over positions touching the current position the player is simulating. The second was taken by calculating all possible moves our player can make from its current position. This was done using BFS. In order for our heuristics to be of equal weight every turn we normalized them on a percentage bases. We would divide (all possible moves / total board size)*100 and again we would represent total walls as ((4 - wallcount)/4)*100. We chose 4 - wallcount as the more walls the worse off we are.

1.2.2 Evaluation Function - Wall Placement

Our evaluation function for our walls were quite similar. We looped through all available wall placements in the current position then checked how many options the hero and villain would have. Then subtract the two values and if it was higher than what we previously had, we update to that wall position. A straightforward approach that accounts for both players in the game.

1.2.3 MiniMax

The minimax function is a key component of the StudentAgent's decision-making process. It's a recursive function that implements the minimax algorithm, which is a decision-making algorithm used in two-player games. The goal is to find the optimal move for the player assuming that the opponent is also playing optimally. The algorithm explores the game tree to evaluate different possible moves and their consequences.

Walkthrough of our minimax function:

Base Case: The function starts with checking whether it has reached a terminal state or the maximum depth (specified by the variable depth = 6) has been reached. If either condition is met, the function returns the evaluation of the current state. The evaluation is done using the evaluate_wall and evaluate function.

Maximizing Player's Turn (is_maximizing_player is True): If it's the maximizing player's turn (in this case, the student agent), the function initializes variables for the maximum evaluation (max_eval), the best move (best_move), and the best wall placement (best_wall_placement). It then iterates through all possible moves and evaluates each move recursively by calling the minimax function with the updated state after the move. The function considers the move with the highest evaluation and updates max_eval, best_move, and best_wall_placement accordingly.

Minimizing Player's Turn (is_maximizing_player is False): If it's the minimizing player's turn, the function initializes variables for the minimum evaluation (min_eval), the best move (best_move), and the best wall placement (best_wall_placement). It iterates through all possible moves of the adversary and evaluates each move recursively by calling the minimax function with the updated state after the adversary's move. The function considers the move with the lowest evaluation and updates min_eval, best_move, and best_wall_placement accordingly.

Returning Results: Finally, the function returns the evaluation value (max_eval or min_eval, depending on whether it's the maximizing or minimizing player's turn), the

best move (best_move), and the best wall placement (best_wall_placement). The algorithm employs iterative deepening, gradually increasing the depth of the search until either the maximum depth is reached or a specified time limit (max_time_seconds) is exceeded. This is done to make the algorithm more responsive to time constraints.

Overall, the minimax function is a critical part of the decision-making process, helping the agent choose the most promising move while considering the opponent's responses.

1.3 Quantitative Performance

1. On a 12x12 board our bot has a depth that ranges anywhere from depth 2-4. for example, if we are in the middle of the board with complete open space, there are way more moves to compute rather than late game where we might be surrounded by several walls. Essentially, as the game progresses we are able to look more moves into the future. We made a simple design to not exclude any moves that are reachable from the hero or villain. We implemented this way as we thought our bot could possibly ignore some very important moves along the way. The result of this is that all branches have the same length unless the timer runs out.

2. The breadth of the min and max player are the same as we consider all possible moves. This was especially important for the min player as even though we are minimizing based on our evaluation function, other teams use different ones so we felt to include all the moves would give a more accurate representation of what the villain may pick as a move. We have a function in our code:

$$get\_all\_possible\_moves(chess\_board, my\_pos, max\_step, adv\_pos)$$

This uses BFS to find all possible moves of my_pos. when maximizing we set my_pos to hero position and when minimizing vice versa.

3. Combining breadth and depth explained in the above paragraphs, out runtime is:

$$O(b^d)$$

where b is the branching factor and d is the depth. This grows exponentially as the tree grows so as the board size grows, branching factor increases along with the depth. As mentioned, a way to prevent exponential run times was to implement a depth limiting search to essentially cap d. Although, we didn't implement a cap on the branching factor b.

4. As mentioned in Section 1.2.1,Our evaluation function for a move consists of two heuristics.

The first heuristic, has essentially no impact on run time and no effect on the breadth and depth achievable. This is because it is a simple process to check the 4 potential positions a wall can be in surrounding the spot on the board. The second heuristic requires the function, explained in 1.3 Q2, to be called. This is a costly procedure, but a necessity for our strategy. Without this step, the breadth and depth achievable could be extended however, our game plan would have to change entirely.

5. Win rate most definitely differs depending on our opponent. After running several thousand simulations vs the random agent, our win rate estimation vs the random

agent will be the most accurate. We believe it will range from 99-100%. It is so high as the random agent essentially traps its self over 50% of the time. A defensive strategy is perfect for this as the longer we wait the higher chance the bot has of trapping its self.

Although, relative to a human agent, we predict our win rate will be much lower than 50%. Our bot doesn't really have sophisticated trapping mechanisms. On the other hand, by continuously moving to open positions, when we win it will be most likely to a human who didn't realize they are in an area with less squares around them and aren't able to leave it in time.

As mentioned in our Motivation, our main thought process was wait for the villain to mess up. This is why we believe against the average bot, we will beat 80-100% as they will mess up quite often. Although, against the top 10% of players we think we will win 30-50%. This should place us in the top 40% of players in the field.

2 Review of our approach

2.1 Advantages and Disadvantages

The defensive approach we have taken has strong advantages. Our method takes advantage of AIs whose heuristics may be faulty or cause the player to be in overly stretched positions. Because ours maintains distance unless a glaring advantage is observed, it also performs well against players who are only attempting to chase the opposition. Unless the opponent can be readily trapped, the emphasis is always on going into the most free space possible.

Since time to make a move is limited, We monitor the duration of the iterative deepening process using a timer. We make an effort to reach the maximum depth, but the process ends automatically just before 2 seconds. This can lead to inconsistent performance across board sizes, but it is also a beneficial thing because it ensures that there won't be a time limit penalty for any move.

A negative to our approach is our player may miss moves that can setup attacks against the opponent. It never engages in offensive play until it can trap the opposition in a single move. Its efficacy against players with excellent assessment functions and accurate decision making may be hampered by a lack of offensive moves.

By using minimax, we are making the assumption that the opponent and our bot are both playing optimally. This concept is sound, however it's important to highlight that it is viewing optimally with respect to the same evaluation function for both players. Since it is improbable that every opponent we face to have the same evaluation function, we might be incorrectly predicting some of their decisions for moves. This could be problematic if an opponent has a significantly different evaluation function and their function is strong.

2.2 Alternative Approaches

When starting this project, we took into account 2 different approaches:

Monte Carlo Tree Search - We felt this would be a better approach in certain situ-

ations. For example, MCT is particularly better for a higher amount of uncertainty and deals better with higher branching factor. Because of this, it does significantly better without an evaluation function. In addition, MCT focuses primarily on the most promising nodes (part of game tree). This allows the algorithm to explore more deeply in areas that have a higher likelihood of winning. Although, we also noticed that because of this feature, there is still a chance that MCT would miss crucial moves if it thinks it is better to go somewhere else. Also, it is much more computationally intensive and with a time constraint we knew it wouldn't be as effective as minimax

Heuristic only - This was the first bot we created. We felt as if our heuristics were strong enough, we would be able to beat much more than 50% of the competition. As it turned out, we realized the more heuristics we added, the more diluted each one was worth. We figured, if we were able to look several moves in advance we wouldn't have to put an extreme amount of pressure on how good our evaluation function would have to be. The heuristic only approach is significantly more computationally efficient as the most intensive part would be running BFS to find all possible moves. It is much much simple than minimax and also relies much more on real time decision making which can be advantageous for a problem similar to ours. On the other hand, it has a limited depth of search, provides a lack of guarantees and most importantly, we are always as strong as our heuristic. Meaning a bad heuristic means a bad student agent.

2.3 Improvements

Although we believe that our bot will beat the average bot, there are a few ideas we had that were not implemented that could of increased its potential performance.

A possible improvement we thought of was changing play style and decision making based on the stage of the game. Through watching testing rounds, it is clear that early, middle and late game performances could benefit from different strategies. The early and middle phase of the game there is a stronger benefit towards a defensive approach while late game is where attacking the opponent makes the most sense. This is because at the beginning of the game establishing the most open space is ideal while at the end you can take advantage of the numerous walls placed on the board and potential trapping goals. Our bot does not take this into account.

In terms of algorithm performance, our implementation of minimax could be improved with alpha-beta pruning. It is likely that implementing alpha-beta pruning would rule out a lot of possible moves quickly. The benefit from this would be increasing the depth that is reached when evaluating and determining what move to make. The depth reached is directly limited by the time frame implemented so clearly increasing the speed of minimax would benefit performance